Parallel Copy Elimination on Data Dependence Graphs

Florian Brandner, Quentin Colombet

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parallel Copy Elimination on Data Dependence Graphs

Florian Brandner — Quentin Colombet

N° 7735

2011

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7735--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Parallel Copy Elimination on Data Dependence Graphs

Florian Brandner* , Quentin Colombet*

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet COMPSYS

Rapport de recherche n° 7735 — 2011 — 29 pages

Abstract: Register allocation regained much interest in recent years due to the development of *decoupled* strategies that split the problem into separate phases: spilling, register assignment, and copy elimination.

Traditional approaches to copy elimination during register allocation are based on interference graphs and *register coalescing*. Variables are represented as nodes in a graph, which are coalesced, if they can be assigned the same register. However, decoupled approaches strive to avoid interference graphs and thus often resort to local *recoloring*.

A common assumption of existing coalescing and recoloring approaches is that the original ordering of the instructions in the program is *not* changed. This work presents an extension of a local recoloring technique called *Parallel Copy Motion*. We perform code motion on data dependence graphs in order to eliminate useless copies and reorder instructions, while at the same time a valid register assignment is preserved. Our results show that even after traditional register allocation with coalescing our technique is able to eliminate an additional 3% (up to 9%) of the remaining copies and reduce the weighted costs of register copies by up to 25% for the SPECINT 2000 benchmarks. In comparison to Parallel Copy Motion, our technique removes 11% (up to 20%) more copies and up to 39% more of the copy costs.

Key-words: Parallel Copy Motion, Data Dependence Graph, Register Allocation, Register Coalescing, Copy Elimination

This work is partly supported by the compilation group of STMicroelectronics.

* Compsys, LIP, UMR 5668 CNRS, INRIA, ENS de Lyon, UCB-Lyon

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Résumé :

L'allocation de registres a eu un regain d'intérêt ces dernières années grâce au développement de stratégies dite découplées qui décomposent le problème en plusieurs phases distinctes: l'éviction en mémoire (spill), l'affectation aux registres et l'élimination de copie.

Les approches traditionnelles pour l'élimination de copie pendant l'allocation de registres sont basées sur les graphes d'interférence et la fusion de noeud (coalescing). Les variables sont représentées par des noeuds dans le graphe qui sont fusionnés si celles-ci peuvent être allouées au même registre. Cependant, les approches découplées s'efforcent d'éviter l'utilisation des graphes d'interférence et ont de fait souvent recours à du recoloriage local.

Une simplification courante dans les approches de coalescing et de recoloriage existantes est que l'ordonnancement des instructions du programme n'est pas remis en cause. Ce travail présente une extension du *Parallel Copy Motion*, qui est une technique de recoloriage local. Nous effectuons des déplacements de code sur le graphe de dépendance de données pour éliminer les copies inutiles et réordonner les instructions, tout en préservant une allocation de registres valide. Nos résultats montrent que même après une allocation de registres traditionnelle avec coalescing, notre technique est capable d'éliminer encore 3% (et jusqu'à 9%) des copies restantes et de réduire le coût pondéré de ces copies jusqu'à 25% pour les benchmarks SPECINT 2000. Par rapport au Parallel Copy Motion, notre technique élimine 11% (et jusqu'à 20%) de copies en plus et jusqu'à 39% en plus du coût de ces copies.

Mots-clés : Parallel Copy Motion, graphe de dépendance de données, allocation de registres, coalescing, élimination de copie

1 Introduction

A major phase of an optimizing compiler is register allocation, which assigns registers to program variables wherever possible. However, if too many variables are *live* at the same time, i.e., carry a value that might eventually be used, the available registers might not be enough to store all those values. In this case some variables have to be assigned to memory, i.e., *spilled*, and, depending on the architecture, additional load and store operations have to be inserted.

A standard technique for register allocation is based on graph coloring [10]. This heuristic approach iteratively tries to color the *interference graph* (IG), where nodes represent the variables of the program. If the coloring fails, a variable is spilled to memory, load and store operations are inserted, and another attempt to color an updated IG is made. Register-to-register copies are eliminated by *coalescing* [10, 7, 15, 20], where variables that are copy-related are merged in the interference graph and thus assigned the same color.

In recent years, so-called *decoupled* approaches to register allocation [1] gained much interest. The basic idea is to split the register allocation problem into separate phases: spilling, register assignment, and copy elimination. The first phase reduces the register demand by spilling variables to memory and ensures that *some* register assignment is viable. The second phase assigns registers to the variables, without further spilling. All decoupled approaches rely on a form of *live-range* splitting to ensure that the assignment succeeds by cutting the original live-ranges of variables into smaller pieces using copy operations. Static Single Assignment (SSA) form [11] provides the basis for live-range splitting in many decoupled approaches. Due to the additional copies, the final copy elimination phase becomes crucial.

Traditional copy elimination by coalescing using IGs generally performs well in this setting [1]. However, recent SSA-based spilling and assignment heuristics [5, 6] avoid the costly construction of IGs, it is thus essential to develop new copy elimination techniques. One solution is to bias the register assignment to heuristically assign the same register to copy-related variables [8, 6]. Another possibility is to perform local recoloring [17, 3] after the assignment phase.

A common limitation of existing approaches to copy elimination is that the ordering of the instructions in the program is *not* modified. We present an extension of a local recoloring technique [3] that operates on *Data Dependence Graphs* (DDG) and eliminates copies by performing local code motion. Our approach is based on parallel copies (see Section 2) that originate from mismatching register assignments at split points. The parallel copies are represented within a DDG, along with all other operations of a basic block. We then perform *upward* and *downward* code motion of instructions reading or defining a register of a parallel copy respectively. The goal is to render this particular register *dead* before or after the parallel copy, i.e. the register's value is no longer used. Once the register is dead, the parallel copy becomes (partially) useless and can be split or even completely eliminated. Not all code motions are permissible. It has to be ensured that all data dependencies are preserved and no values are lost. In particular, we have to ensure that no cyclic dependencies are introduced in the DDG, which prevent an ordering of the instructions after copy elimination.

The main contribution of this article is to show that the data dependence graph can be kept consistent using rather simple and elegant transformation rules that split and eliminate parallel copies until no further simplification is possible or a predefined threshold is reached.

The remainder of this paper is organized as follows. We give some background on notations, data dependence graphs, and parallel copies in Section 2. Next, we will describe our approach to copy elimination on data dependence graphs in Section 3. Section 4 presents details on experiments conducted using the SPECINT 2000 benchmark suite. Related work is presented in Section 5 before concluding in Section 6.

2 Background

2.1 Data Dependence Graphs

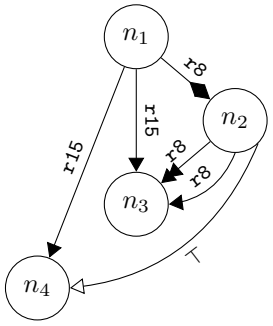
The presented algorithms operate on data dependence graphs to eliminate copies, we thus give a brief definition:

Definition 1. A *Data Dependence Graph* (DDG) is an acyclic graph $G = (V, E, L)$, where nodes n in V represent instructions and labeled edges (u, v, l) in $E \subseteq V \times V \times L$ dependencies among instructions. We distinguish four kinds of labels in L: (1) true register dependencies $\overleftarrow{r}$, (2) register output dependencies $\overleftrightarrow{r}$, (3) register anti dependencies $\overrightarrow{r}$, where r is a register name, and, finally, (4) other dependencies $\top$.

Data dependence graphs are best represented using a graphical notation as depicted by Figure 1, showing a linear code fragment and its DDG. Throughout this paper we follow the convention that true dependencies are represented by an arrow with a filled triangle tip (→), output dependencies by an arrow with two overlapping triangles (↠), and anti dependencies by an arrow with a diamond tip (⇾). Other dependencies are represented by an open triangle (⇢).

```
1:  mov r15 = r8;
2:  ldw r8 = 44[r9];
3:  add r8 = r8, r15;
4:  stw 44[r9] = r15;
```

(a) Linear Assembly Code



(b) Data Dependence Graph

Figure 1: Assembly code (a) and the corresponding data dependence graph (b).

2.2 Parallel Copies

We rely on the notion of *parallel copies* to represent, for every split point, the set of mismatching register assignments and a set of atomic copy operations to

fix-up those mismatches. Under SSA form, for instance, parallel copies originate from ϕ-operations where the register allocator was not able to assign the same register to some of its source and destination operands [18].

Definition 2. A *Parallel Copy* is a set of register-to-register copies $(d_1, \ldots, d_n) := (a_1, \ldots, a_n)$, where $\forall i, j,\ i \neq j : d_i \neq d_j$, that are executed in parallel, i.e., all argument registers a_i, $1 \leq i \leq n$, are first read and all destination registers d_i, $1 \leq i \leq n$, are subsequently written at the same time. We call a parallel copy *regular* if $d_i = a_{i+1}$, $1 \leq i < n$, and $\forall i : d_i \neq a_1$. A parallel copy is said to be *cyclic* if it is regular except that $d_n = a_1$.

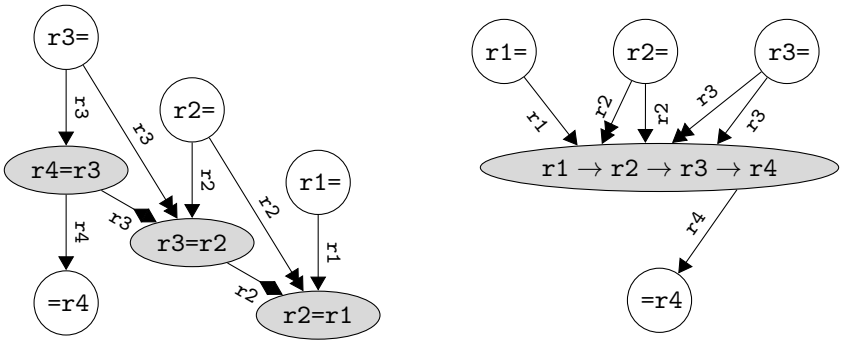
A regular parallel copy $(d_1, \ldots, d_n) := (a_1, \ldots, a_n)$ can be represented conveniently as a chain $a_1 \to d_1 \to \ldots \to d_n$. Cyclic copies on the other hand form a closed loop, as shown below:

$$\circlearrowright d_1 \to \ldots \to d_n \quad .$$

Using more general graph structures, other forms of parallel copies can be defined. The most general case is represented by graphs where the nodes have an in-degree of at most 1, i.e., each register is defined at most once. This form of parallel copies allows the *duplication* of register values. For the remainder of this work, we consider regular or cyclic copies only, more complex graphs are assumed to be decomposed beforehand [2]. The algorithms presented in the following are, in principle, applicable to more general graph structures under minor modifications.

When parallel copies are represented in a DDG, we implicitly assume a set of individual register-to-register copies that are merged into a single node. The dependencies between those moves and other instructions in the DDG are directed to the merged node accordingly. Representing parallel copies as a single node has the advantage that the resulting DDG is free of cycles, which would arise in the case of cyclic copies.[1] Figure 2 shows the two ways of representing

[1] Cyclic parallel copies could also be represented as a sequence of swap instructions. This would avoid the cycles in the DDG, but would complicate the copy elimination presented later.



(a) DDG with a set of copy nodes

(b) DDG with a merged parallel copy node

Figure 2: A parallel copy `r1` → `r2` → `r3` → `r4` represented as a set of register-to-register copies (a) and as a single parallel copy (b) in a DDG.

a parallel copy in a DDG: once using explicit moves and once using a merged copy node. Note that all dependencies are reflected equally in both versions. in particular, the anti-dependencies among the individual copies are captured by the copy chain `r1` $\rightarrow$ `r2` $\rightarrow$ `r3` $\rightarrow$ `r4` within the parallel copy node.

2.3 Parallel Copy Motion

Our approach is an extension of the *Parallel Copy Motion* technique of Bouchez et al. [3]. In contrast to what the name suggests, parallel copy motion operates on register permutations covering all registers of the processor. The parallel copies are thus turned into permutations before the algorithm starts. The problem is then to find a good placement of these permutations within the program to minimize (1) the number of copies arising from projecting those permutations on the live registers at their final position and (2) the execution frequency of those copies, where the frequencies are either based on static estimates or profiling feedback.

Their algorithm proceeds by first treating permutations on critical edges within the control flow graph. This *Edge Motion* phase tries to use permutations on neighboring critical edges to cancel each other out and reduce the execution frequency of the resulting permutations. At the end, all permutations are either assigned to a basic block or otherwise the respective critical edge is split by forming a new basic block, the copy is then assigned to this block.

Next, during the *Block Motion* phase, the permutations are placed within basic blocks. The algorithm again tries to combine permutations to cancel each other out. The permutations are then placed within their basic blocks such that the number of copies, induced by projecting them to the set of live registers, is minimized using liveness information.

Basically, we reuse the *Edge Motion* phase without modification and replace the *Block Motion* phase by a more powerful technique based on data dependence graphs as explained in the next section.

3 Copy Elimination on Data Dependence Graphs

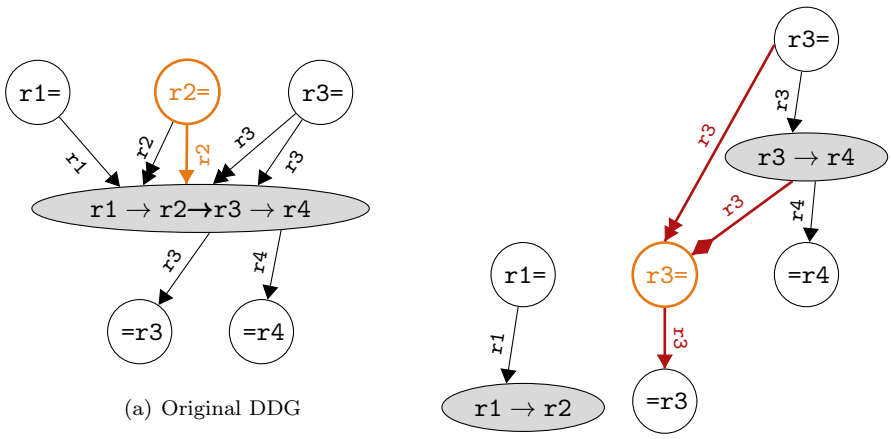
It is easy to see that it is possible to (partially) eliminate parallel copies by transforming the DDG and renaming of register operands. We will present two different transformations that implicitly reorder the instructions of a basic block in order to eliminate parallel copies. The main idea is to *move* instructions within the DDG *upward* or *downward* past the parallel copy, while at the same time renaming register operands. The involved parallel copy is split into smaller pieces as a side-effect of these transformations. This, for one, eliminates useless copies. In addition, it might enable other copy eliminations and break cyclic parallel copies, which have to be realized using more costly swap operations.

3.1 Downward Motion of Definitions

The first form of transformation is to perform a *downward* motion of a *definition* of a register that is *used* by a parallel copy, i.e., the DDG contains a true dependence between the definition and the copy. It is then possible to move the definition down past the parallel copy while replacing the original register of the

definition with the corresponding destination register of the parallel copy. The argument of the parallel copy becomes *dead* as a side-effect of this transformation, since the definition previously supplying the value now follows the parallel copy in any valid linear ordering of the DDG. The respective register-to-register copy thus becomes useless and can be eliminated, i.e., the parallel copy is split into two pieces. Due to the register renaming and splitting of the parallel copy, the DDG might need some additional updates in order to ensure correctness.

A more formal algorithm will be presented below, but we will first give a short example. Consider the original DDG from Figure 3(a). The value calculated for register `r2`, the DDG node representing the definition is highlighted in orange, is immediately copied to register `r3`, without any other instruction touching the value. It is easy to see that this copy operation can be avoided by register renaming and a minor update of the DDG. Figure 3(b) shows the final DDG after performing this transformation. Due to the renaming, some additional dependencies have to be added to the DDG as highlighted in red. It is important to note that these dependencies can be easily derived from the dependencies of the original DDG node of the parallel copy. We also see that the copy has been split into two smaller pieces `r1` → `r2` and `r3` → `r4`, while the copy `r2` → `r3` was eliminated. This splitting is particularly interesting to break cyclic parallel copies because register swaps can be avoided.



(a) Original DDG

(b) Transformed DDG

Figure 3: A DDG before (a) and after (b) performing a downward motion of the definition of register `r2`.

3.1.1 Handling Regular Parallel Copies

Algorithm 1 shows the main steps required to perform a downward motion of an instruction, denoted *def*, defining a register r with respect to a regular parallel copy operation *copy*. The algorithm first verifies that a linear ordering can be derived from the DDG after performing the transformation, i.e., it verifies that no dependence cycles are introduced – see lines 1–6. The transformation is not performed if this check fails. The transformation itself consists of (1) renaming the destination register of the definition, (2) splitting the parallel copy, and (3) updating the DDG – see lines 7–13. It is important to note that

Algorithm 1 Perform downward motion of a definition.

```
DefMotionDown(DDG G = (V, E), DDGNode def, Register r,
              DDGNode copy)
1  // Ensure that no other uses exist besides copy
2  uses = {e | e = (def, u, r←) ∈ E, u ∈ V}
3  if uses ≠ {(def, copy, r←)}
4      return
5  // Check dependencies and transform the DDG
6  if ¬ExistsPathFromDef(G, def, r, copy)
7      // Rename the destination register of def
8      u = ResultOfArgument(copy, r)
9      RenameResult(def, r, u)
10     // Split the parallel copy
11     (lcopy, rcopy) = SplitAtArgument(G, copy, r)
12     // Update the data dependencies
13     UpdateDefDependencies(G, def, r, u, lcopy, rcopy)
```

only the destination register of the definition is renamed during this processing, all other registers, in particular, other register uses are *not* modified. Before we describe the phases in more detail, we define a few helper functions required during the processing:

- ArgIndex and ResIndex take a parallel copy and a register as arguments and return the index within the argument list of the parallel copy. For instance, given the parallel copy $c = (d_1, \ldots, d_n) := (a_1, \ldots, a_n)$, ArgIndex$(c, a_i)$ will return i, while ResIndex(c, d_j) will return j.
- ResultOfArgument takes a parallel copy and a register as an argument and returns the respective destination register of the argument register, i.e., for a copy $c = (d_1, \ldots, d_n) := (a_1, \ldots, a_n)$ and register a_i, ResultOfArgument(c, a_i) returns d_i.
- The function RenameResult performs a simple renaming of the destination registers of an instruction represented by a DDG node.

Preventing Cyclic Dependencies

In order to perform a downward motion of a definition, it has to be assured that all data dependencies can be satisfied while a linear order of the DDG can still be derived after performing the transformation. Two situations have to be considered: (1) uses of the register defined by the instruction other than the parallel copy and (2) dependencies between the definition and the parallel copy.

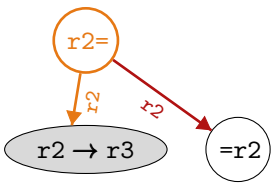
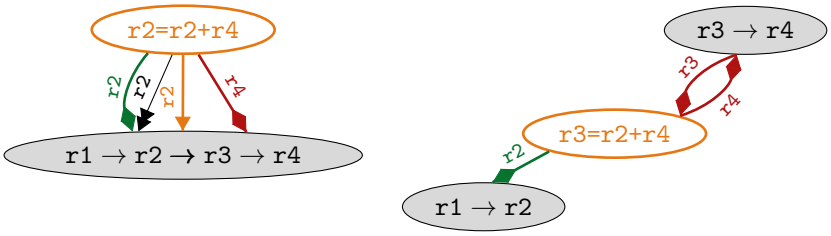


Figure 4: Care has to be taken that dependencies between a definition and its uses are not lost.



(a) Original DDG (b) Transformed DDG

Figure 5: Performing a downward motion on the DDG in (a) results in cyclic data dependencies after the transformation (b).

In the former case, other uses than the parallel copy are simply rejected. This is foremost a simplification of the algorithm in order to avoid the need to track register uses and their relative position to the parallel copy in question. In particular, this avoids the need to track uses that are otherwise independent from the parallel copy, as shown by Figure 4. Note that we can always choose to rename those uses beforehand to allow the downward motion of the definition.

The second case arises foremost from data dependencies between the parallel copy and the definition, e.g., when the parallel copy uses or defines a register operand of the instruction *def*. In this case, after updating the DDG, we might encounter that the parallel copy has a (transitive) dependence leading to the definition, while at the same time a (transitive) dependence from the definition leads back to the parallel copy.

Figure 5 shows such an example. The anti dependence labeled with `r4` and marked red in Figure 5(a) leads to a cyclic dependence after the transformation. Note that it would be possible to resolve this issue by additional renaming if the respective values are still accessible through other registers. However, in

Algorithm 2 Check (transitive) dependencies between an instruction defining a register and a regular parallel copy using the same register.

ExistsPathFromDef(*DDG* $G = (V, E)$, *DDGNode* *def*, *Register* r, *DDGNode* *copy*)

```
1  foreach (n, copy, l) ∈ E where a path def →* n exists in G
2      if n = def ∧ l ∈ {←r, →r, ↔r}
3          // Ignore all direct register dependencies
4      elseif ∃r_d : l ∈ {↔r_d, →r_d}
5          // Ignore anti and output dependencies to the parallel
6          // copy if the involved operand appears before register r
7          if ArgIndex(copy, r) ≤ ResIndex(copy, r_d)
8              return TRUE
9      elseif ∃r_u : l = ←r_u
10         // Ignore true dependencies to the parallel copy if the
11         // involved operand appears before register r
12         if ArgIndex(copy, r) < ArgIndex(copy, r_u)
13             return TRUE
14     else
15         return TRUE
16 return FALSE
```

the given example this is not possible since the value of `r4` is destroyed. We will come back to this issue in Section 3.3. The algorithm presented here does not consider the possibility of additional renaming as practical examples rarely require these transformations.

This is, in part, due to the fact that not all dependencies between the copy and the definition immediately lead to cyclic dependencies. Consider, for instance, the anti dependence labeled with `r2` and marked green in Figure 5(a). This dependence remains in the final DDG even after the transformation is applied *without* causing any cycles – see Figure 5(b). The reason for this is that the definition of register `r2` appears *before* the point where the parallel copy is split. The dependence thus cannot, in any case, cause a cyclic dependence. This property applies to all register dependencies leading to regular parallel copies and will be exploited in the following.

Algorithm 2 explores all paths in the DDG leading from the definition *def* defining register r to the parallel copy *copy* using that register. It verifies whether the respective dependence can or cannot lead to a cycle when the definition is moved downwards past the parallel copy. The algorithm distinguishes four cases:

1. All direct register dependencies between the definition and the parallel copy carrying the definition's register can safely be discarded – see line 2. True and output dependencies are automatically resolved by renaming the destination register of the definition, i.e., the corresponding dependencies disappear. Anti dependencies, on the other hand, may remain. However, due to the splitting of the parallel copy this is not an issue.
2. All anti and output dependencies can be ignored if the register which labels the dependence (denoted as r_d in the algorithm) appears before the definition's register in the parallel copy – see line 4. The respective dependencies will remain in the final DDG, however, due to the splitting of the parallel copy it is ensured that no cycles can arise.
3. Finally, all true dependencies can be ignored using a similar argument – see line 9.
4. All other dependencies, in particular, non-register dependencies, are treated conservatively – see line 14.

The usage of the relative position of arguments of the parallel copy is best understood when viewing the parallel copy as a chain of individual copy operations – as for example shown in Figure 2. Splitting the parallel copy then corresponds to splitting this chain of copy operations. Dependencies leading to the head of the chain, i.e., before the actual split point, are problematic and will cause cyclic dependencies, while those leading to the tail of the chain, i.e., after the split point, are safe. The chain of copies can be constructed from a regular parallel copy by processing the arguments of the copy in *reverse* order.

Cyclic parallel copies, however, do not form a chain of copies but a cycle and thus need special treatment. We will return to this problem at the end of this section after discussing the final update procedure to keep the DDG in a consistent state after a downward motion of a definition.

Transforming the Dependence Graph

The transformation phase of downward code motion consists of three steps. First, the destination register of the definition is renamed – see Algorithm 1

line 9. This step is straightforward it is not further discussed here. We will instead focus on the splitting of the parallel copy (line 11) and the update of the DDG (line 13).

The parallel copy is split at the use point of the definition's register during the transformation using SplitAtArgument, resulting in two independent parallel copies. Assume a parallel copy $c = (d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_n) := (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ and a register $r = a_i$. The splitting gives two copies $(d_1, \ldots, d_{i-1}) := (a_1, \ldots, a_{i-1})$ and $(d_{i+1}, \ldots, d_n) := (a_{i+1}, \ldots, a_n)$, where the first is referred to as *lcopy* and the second as *rcopy*. We, further, assume that, by splitting the copy, corresponding nodes are added to the DDG. The DDG nodes inherit the respective dependencies from the DDG node of the original parallel copy, which is discarded after the splitting. Note that both, *lcopy* and *rcopy*, may be empty at this point. For the sake of simplicity we assume that an empty copy will be inserted to the DDG in such a case – empty copies can easily be eliminated by a post-processing pass.

After splitting the parallel copy the DDG has to be updated in order to account for new and/or eliminated data dependencies due to the register renaming, the copy elimination induced by the splitting of the parallel copy, or the code motion of the definition. Algorithm 3 shows the necessary steps to update the DDG given a definition *def* defining a register r, which is to be updated by register u, and two split pieces of the original parallel copy *lcopy* and *rcopy*.

Algorithm 3 Update the dependencies of the DDG after a downward copy motion of a definition.

UpdateDefDependencies(*DDG* $G = (V, E)$, *DDGNode def*, *Register* r, u, *DDGNode lcopy*, *rcopy*)

1 // Account for redefinition of u by *def*
2 add $(rcopy, def, \overrightarrow{u})$ to E
3 // Remove spurious dependencies
4 remove $(def, lcopy, l)$ where $l \in \{\overleftrightarrow{r}, \overleftarrow{r}\}$ from E
5 remove $(def, rcopy, \overrightarrow{u})$ from E
6 remove $(rcopy, n, \overrightarrow{u})$ from E, for all $n \in V$
7 // Transfer output and anti dependencies between *def* and *lcopy*
8 **foreach** $e = (n, def, l) \in E$ where $l \in \{\overleftrightarrow{r}, \overrightarrow{r}\}$
9 remove e from E
10 add $(n, lcopy, l)$ to E
11 // Transfer output and true dependencies between *rcopy* and *def*
12 **foreach** $e = (rcopy, n, l) \in E$ where $l \in \{\overleftrightarrow{u}, \overleftarrow{u}\}$
13 remove e from E
14 add (def, n, l) to E
15 // Transfer output and anti dependencies between *rcopy* and *def*
16 **foreach** $e = (n, rcopy, l) \in E$ where $l \in \{\overleftrightarrow{u}, \overrightarrow{u}\}$
17 remove e from E
18 add (n, def, l) to E
19 // Create an output dependence to the next definition of u
20 **if** $\exists (def, n, \overleftrightarrow{u}) \in E$ and def has a use of u
21 add $(def, n, \overrightarrow{u})$ to E

In a first step, see line 2, a new mandatory data dependence between the right piece of the parallel copy and the definition is added to the DDG labeled with u, i.e., the new destination register of the definition. Next, dependencies are removed that are superfluous (line 4), either due to the register renaming at the definition or the copy elimination.

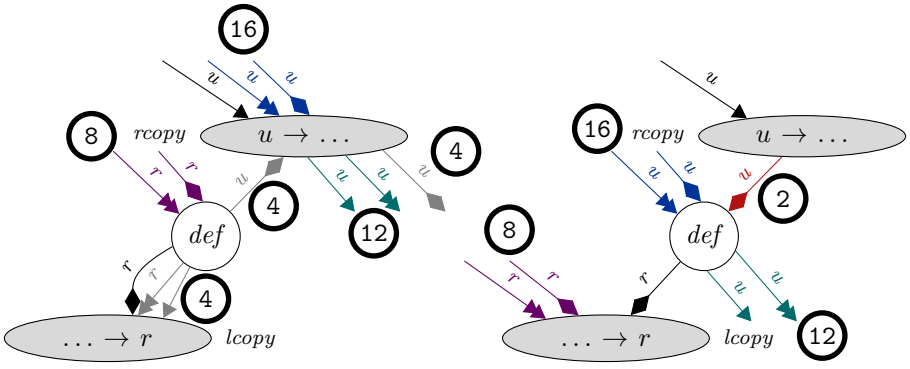
During the next step (line 8) anti and output dependencies leading to the definition and labeled with the definition's original register are *redirected* to the left piece of the parallel copy. Due to the register renaming these dependencies cannot point to the definition anymore and have to be redirected. The left piece of the copy is known to be either empty or is known to end with an assignment to the original register of the definition. The dependencies thus have to be redirected to the left parallel copy to preserve correctness. In the case of an empty parallel copy the dependencies might later turn out to be superfluous or have to be further redirected to reach the closest actual assignment to the respective register. Note, true dependencies are not affected by the renaming and thus are left untouched.

A similar transfer of dependencies is performed for output and true dependencies originating from the right piece of the parallel copy and labeled with the definition's new destination register u – see line 12. Due to the renaming and the code motion of the definition these dependencies have to be redirected in order to correctly capture the new position and the new destination register of the definition.

Yet another transfer (line 16) of dependencies is performed for dependencies leading to the right half of the parallel copy and labeled with the definition's new destination register. The dependencies have to be redirected due to the register renaming at the definition and the copy elimination induced by the splitting of the parallel copy.

In a final step (line 20) a potential anti dependence carrying the new destination register of the definition is added to the DDG if the definition uses that register. Note that a corresponding dependence originating at the right half of the parallel copy was removed at line 6.

Figure 6 illustrates the various steps performed during the DDG update labeled with the corresponding line number of Algorithm 3. The affected data



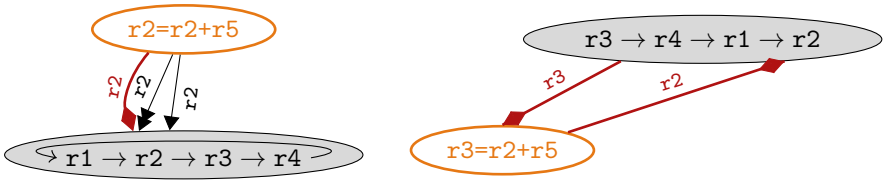
(a) Before transforming the DDG (b) After transforming the DDG

Figure 6: Illustration of the DDG update procedure after splitting the parallel copy, showing the DDG before (a) and after (b) the transformation. The bold circled numbers indicate the corresponding line of Algorithm 3.

dependencies are, in addition, color-coded: dependencies that remain untouched are shown in black, those that are removed in gray, newly added dependencies are indicated in red, while other dependencies transferred between the parallel copies and the definition are shown in matching colors at their original and final position before and after the transformation (violet, blue, green).

3.1.2 Handling Cyclic Parallel Copies

Cyclic copies have to be treated conservatively. If a path exists in the DDG that leads from the definition to the parallel copy, a cyclic dependence will be introduced, unless the path is of length 1 consisting of a single true or output dependence labeled with the definition's register. These direct dependencies can safely be ignored since the parallel copy will be split and the respective dependencies will be removed during the transformation. Note that anti dependencies cannot be discarded at this point, unless the respective register use is renamed – which we do not considered here. Figure 7 shows an example of this situation.



(a) Original DDG (b) Transformed DDG

Figure 7: Performing a downward motion on the DDG in (a) results in cyclic data dependencies after the transformation (b).

The algorithms presented previously for the case of regular parallel copies need to be adapted in order to handle cyclic parallel copies correctly. Firstly, splitting a parallel copy never gives two non-empty regular parallel copies, but only one. The function SplitAtArgument is thus redefined in the case of a cyclic parallel copy as follows: given a cyclic parallel copy

Algorithm 4 Check (transitive) dependencies between an instruction defining a register and a, potentially cyclic, parallel copy using the same register.

```
ExistsPathFromDef(DDG G = (V, E), DDGNode def, Register r,
                  DDGNode copy)
1  foreach (n, copy, l) ∈ E where a path def →* n exists in G
2      if cyclic
3          // Ignore direct true and anti dependencies for cyclic parallel copies
4          if n ≠ def ∨ l ∉ {←r, ↔r}
5              return TRUE
6      else
7          // Code of Algorithm 2
8          ...
9  return FALSE
```

Algorithm 5 Update the dependencies of the DDG after a downward copy motion of a definition for potentially cyclic parallel copies.

UPDATEDEFDEPENDENCIES(*DDG* $G = (V, E)$, *DDGNode* *def*, *Register* r, *u* *DDGNode* *lcopy*, *rcopy*)

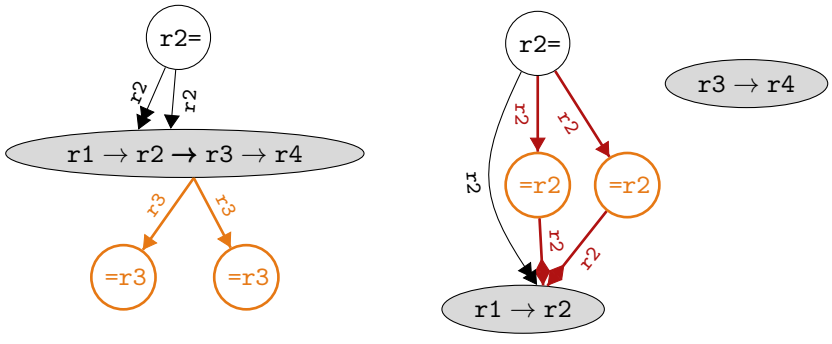
```
1  // Code of Algorithm 3
2  ...
3  if cyclic
4      foreach e = (n, rcopy, l) ∈ E where l ∈ {↔r, →r}
5          remove e from E
6          add (n, def, l) to E
```

$c = (d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_n) := (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ and a register $r = a_i$. The function returns an empty parallel copy and a non-empty copy $(d_{i+1}, \ldots, d_n, d_1, \ldots, d_{i-1}) := (a_{i+1}, \ldots, a_n, a_1, \ldots, a_{i-1})$, i.e., the non-empty copy is denoted as *rcopy* throughout the algorithms.

A second issue stems from the fact that a cyclic parallel copy defines and uses every operand register once. Note, in particular, that the original destination register of the definition might be redefined by the parallel copy. This causes additional dependencies that have to be accounted for in order to prevent cyclic dependencies. Algorithm 4 shows an adapted variant of the EXISTSPATHFROMDEF function. Also the update procedure needs a minor adaption as shown by Algorithm 5.

3.2 Upward Motion of Uses

Another form of transformation is to perform an *upward* code motion of *all* uses of a register defined by a parallel copy while renaming the respective register uses. The result of this transformation, as before, is that the involved register becomes dead, the copy operation can thus be eliminated and the parallel copy split. As before, we start by giving an informal example of the transformation and then present detailed algorithms.



(a) Original DDG (b) Transformed DDG

Figure 8: A DDG before (a) and after (b) performing an upward motion of all uses of register `r3`.

Consider the DDG from Figure 8(a), where all uses of register `r3`, which is defined by the parallel copy, are highlighted in orange. The copy `r2` → `r3` can be eliminated if all these uses are renamed as shown by Figure 8(b). As with the downward motion of definitions, some dependencies become useless due to this transformation, while at the same time new dependencies arise. For instance, the true dependence of the respective uses have to be updated to reflect the reordering and register renaming, as indicated by the true dependencies in red. In addition, new anti dependencies arise due to the definition of `r2` by the parallel copy (also highlighted in red). Similar to before, the transformation may cause cyclic dependencies, which have to be avoided.

3.2.1 Handling Regular Parallel Copies

In contrast to the code motion of definitions, *all* uses have to be considered during an upward motion. The following algorithm thus operates on the set of all uses, but otherwise follows the same principal phases as the downward motion of a definition. Given a set of DDG nodes *uses* reading register r and a parallel copy *copy* defining the same register, it is first verified that the upward motion is legal and does not cause any cyclic dependencies (lines 1–6). In the next step all register uses are renamed (line 9), before the parallel copy is split (line 11). Finally, the data dependence graph is updated for every use (line 13). The algorithm, and the other algorithms following hereafter, make use of some helper functions, which are defined as follows:

- ArgIndex and ResIndex are defined as before – see Section 3.1.1.
- ArgumentOfResult takes a parallel copy and a register as an argument and returns the corresponding argument of the parallel copy for the matching destination register, i.e., for a copy $c = (d_1, \ldots, d_n) := (a_1, \ldots, a_n)$, ArgumentOfResult$(c, d_i)$ returns a_i.
- The function RenameUses performs a simple renaming of the argument registers of the instructions represented by the DDG nodes given by the second argument.

Algorithm 6 Perform downward motion of a definition.

UseMotionUp(*DDG* $G = (V, E)$, *DDGNodes uses*, *Register* r, *DDGNode copy*)

```
1  // Ensure that no other uses exist
2  all_uses = {e | e = (def, u, r̄) ∈ E, u ∈ V}
3  if all_uses ≠ uses
4      return
5  // Check dependencies and transform the DDG
6  if ¬ExistsPathToUses(G, uses, r, copy)
7      // Perform register renaming for every use
8      u = ArgumentOfResult(copy, r)
9      RenameUses(G, uses, r, u)
10     // Split the parallel copy
11     (lcopy, rcopy) = SplitAtResult(G, copy, r)
12     // Update the data dependencies
13     UpdateUseDependencies(G, uses, r, u, lcopy, rcopy)
```

Preventing Cyclic Dependencies

In order to ensure that the DDG is in a valid state after an upward motion of the uses of a register defined by a parallel copy, it has to be guaranteed that (1) *all* uses are considered, and (2) no cyclic dependencies arise from the transformation.

The former case depends, to some extent, on the DDG representation, in particular, on how register uses outside of the scope of the currently considered DDG are represented. For instance, if the DDG covers basic blocks only, registers might be used by an instruction in a successor basic block, i.e., the register is live-out of the current basic block. The corresponding use is not amenable to code motion, since it is not covered by the DDG. For this work, we assume that artificial DDG nodes represent all *external* uses of registers live-out of the code region covered by the DDG. Since those artificial uses are not amenable to code motion they cannot appear in the set of *uses* in Algorithm 6, but may well appear in the set *all_uses* (line 2).

The second issue is similar to the problem of cyclic dependencies that appeared for the downward motion of definitions. Algorithm 7 shows the corresponding test that verifies that no cyclic data dependencies may arise from the transformation. The test is, in fact, very similar to that of Algorithm 2, except that the direction of the examined paths is inverted (line 2), i.e., DDG edges originating from the parallel copy are examined. Consequently, the relation between anti and true dependencies and the relative position of the respective operands within the parallel copy is inverted too (see line 9 and 4). The algorithm otherwise proceeds in the same manner as the corresponding version for the downward motion of definitions, please refer to Section 3.1.1 for a more detailed discussion.

Algorithm 7 Check (transitive) dependencies between a regular parallel copy defining a register and all uses of that register.

```
ExistsPathToUses(DDG G = (V, E), DDGNodes uses, Register r,
                 DDGNode copy)
 1  foreach (copy, n, l) ∈ E where a path n →* u exists in G, u ∈ uses
 2      if n = u ∧ l ∈ {r←, r→, r↔}
 3          // Ignore all direct register dependencies
 4      elseif ∃r_d : l ∈ {r_d↔, r_d→}
 5          // Ignore true and output dependencies originating from the
 6          // parallel copy if the involved operand appears after register r
 7          if ResIndex(copy, r) > ResIndex(copy, r_d)
 8              return true
 9      elseif ∃r_u : l = r_u→
10          // Ignore anti dependencies originating from the parallel copy
11          // if the involved operand appears after register r
12          if ResIndex(copy, r) ≥ ArgIndex(copy, r_u)
13              return true
14      else
15          return true
16  return false
```

Transforming the Dependence Graph

The transformation phase of the upward code motion of uses consists of three principal steps – see Algorithm 6, line 7–13. First, register renaming is performed using the ArgumentOfResult and RenameUses functions (see line 9).

In the next step, the original parallel copy is split using the SplitAtResult function, which given a parallel copy $c = (d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_n) := (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ and a register $r = d_i$ returns two new parallel copies, $(d_1, \ldots, d_{i-1}) := (a_1, \ldots, a_{i-1})$ and $(d_{i+1}, \ldots, d_n) := (a_{i+1}, \ldots, a_n)$, which are denoted as *lcopy* and *rcopy* respectively. The two copies inherit the respective data dependencies from the original parallel copy – in particular, *rcopy* is assumed to inherit all dependencies carrying the original register and *lcopy* is assumed to inherit all dependencies carrying the new register used for renaming. The main difference to the corresponding function SplitAtArgument from Section 3.1.1 is that the registers defined by the parallel copy are examined instead of the registers used in order to determine the split point. An additional difference will become apparent when the handling of cyclic parallel copies is discussed later in the next section.

Finally, the data dependencies of the DDG, i.e., of the involved uses and pieces of the parallel copy, have to be updated. Since only register uses are touched the situation is simple, only true and anti dependencies originating from respectively leading to the use and the left half of the parallel copy as well as true and output dependencies to respectively from the right half of the copy have to be considered. The DDG update procedure, shown by Algorithm 8,

Algorithm 8 Update the dependencies of the DDG after a downward copy motion of a definition.

UpdateUseDependencies(*DDG* $G = (V, E)$, *DDGNodes uses*, *Register* r, u, *DDGNode lcopy*, *rcopy*)

```
1  // Account for output dependencies after removing the definition of r
2  if ∃n1, n2 ∈ V : (n1, rcopy, ↔r) ∧ (rcopy, n2, ↔r)
3       add (n1, n2, ↔r) to E
4  remove (rcopy, n, ↔r) from E, n ∈ V
5  remove (n, rcopy, ↔r) from E, n ∈ V
6  // Remove spurious dependencies
7  remove (rcopy, n, l) from E, ∀n ∈ uses, where l ∈ {←r, ↔r}
8  remove (n1, n2, →r) from E, ∀n1 ∈ uses, n2 ∈ V
9  // Transfer anti dependencies carrying the new register u
10 if lcopy is not empty
11      add (n, lcopy, →u) to E, ∀n ∈ uses
12      remove (lcopy, n, →u) from E, ∀n ∈ uses
13 elseif ∃n1 ∈ V : (lcopy, n1, →u) ∈ E
14      add (n2, n1, →u) to E, ∀n2 ∈ uses
15      remove (lcopy, n1, →u) from E
16 // Transfer true dependencies carrying the new register u
17 if ∃n1 ∈ V : (n1, lcopy, ←u) ∈ E
18      add (n1, n2, ←u) to E, ∀n2 ∈ uses
19      remove (n1, lcopy, ←u) from E
```

takes a set of DDG nodes *uses*, a register r, which is read by the uses, a register u to update those uses and two split pieces of the original parallel copy *lcopy* and *rcopy*. It proceeds in four main steps as follows.

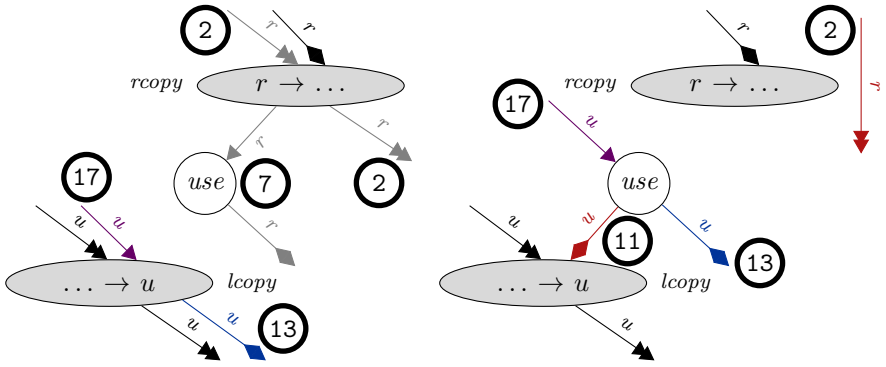
First, output dependencies carrying the original register r are handled (line 2). If *rcopy* has only a single incoming or outgoing output dependence, this dependence is simply removed, since the previous copy defining r has been eliminated by the upward code motion of its uses. However, if *rcopy* has an incoming *and* outgoing output dependence, i.e., r was defined before and after the original parallel copy, then a new dependence has to be added to the DDG establishing a relation between these two definitions.

Next, dependencies carrying the original register r involving any of the uses are removed (line 7). These dependencies became obsolete, due to the renaming of the respective uses. No other action is required involving those dependencies as uses, as opposed to definitions, do not interfere with other uses or definitions succeeding or preceding the respective instructions.

In the subsequent step, output dependencies carrying the register u that has been used during the renaming step are added to the DDG. Two cases need to be distinguished: the left half of the parallel copy *lcopy* is either empty or not, i.e., the copy either redefined u or not. If the register is actually redefined an output dependence leads from all uses to *lcopy*. In case the copy is empty output dependencies are only appended if a definition of u actually followed the original parallel copy, i.e., a corresponding output dependence already existed.

By now the DDG is almost complete, the only missing dependencies are true dependencies carrying the new register u. These can simply be derived from the left half of the parallel copy, if *lcopy* has an incoming true dependence, corresponding dependencies are appended to the DDG leading to every new use created by register renaming.

An illustration of the individual steps of the update procedure are shown by Figure 9. The bold circled numbers relate the respective dependencies with the corresponding code lines of Algorithm 8. The color coding indicates untouched dependencies (black), potentially removed dependencies (gray), newly added dependencies (red), and dependencies transferred between the parallel copy and the uses under consideration (violet/blue).



(a) Before transforming the DDG (b) After transforming the DDG

Figure 9: Illustration of the DDG update procedure, showing the DDG before (a) and after (b) the transformation. The bold circled numbers indicate the corresponding line of Algorithm 8.

3.2.2 Handling Cyclic Parallel Copies

The upward motion of uses is affected by cyclic parallel copies to a lesser extent, since even regular copies might redefine both the uses' register before and after renaming. The DDG update procedure of Algorithm 8 can directly be applied to cyclic copies without modification. However, in order to prevent cyclic data dependencies Algorithm 7 has to be extended. The problem arises from the way cyclic copies are split, we will thus first discuss how the splitting is defined.

As noted before, the splitting of a cyclic parallel copy always yields an empty and a non-empty parallel copy. In Section 3.1.2, for the downward motion of definitions, the function SplitAtArgument was extended accordingly to correctly split cyclic copies, where the non-empty portion was considered as the right half of the parallel copy (denoted *rcopy* in the algorithms). The function SplitAtResult is refined in exactly the same way, this time, however, the non-empty part is considered as the *left* piece, i.e., denoted as *lcopy*.

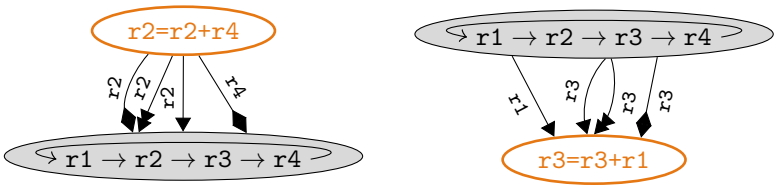
Since the register of the respective uses after renaming is redefined by the parallel copy, it is now easy to see that any dependence between the copy and any use immediately leads to a cycle in the DDG, unless the dependence is a true dependence carrying register r. Algorithm 9 ensures that, in the case of cyclic copies, all other forms of dependencies are rejected when paths between the copy and the respective uses are explored.

Algorithm 9 Check (transitive) dependencies between a potentially cyclic parallel copy defining a register and all uses of that register.

```
ExistsPathToUses(DDG G = (V, E), DDGNodes uses, Register r,
                 DDGNode copy)
1  foreach (copy, n, l) ∈ E where a path n →* u exists in G, u ∈ uses
2      if cyclic
3          // Ignore true dependencies only
4          if l ≠ r←
5              return TRUE
6      else
7          // Code of Algorithm 7.
8          ...
9  return FALSE
```

3.3 Code Motion Past Cyclic Parallel Copies

In the previous sections the focus was on *eliminating* individual copies by moving individual instructions or sets of instructions downward or upward past a regular or cyclic parallel copy. The goal was to render a given register dead by renaming the corresponding definition or all uses respectively. In addition, an instruction can be moved past a cyclic parallel copy, both downward or upward, by renaming *all* register operands of the instruction that appear in the parallel copy. The only requirement is that no (transitive) non-register dependencies exist between the instruction's DDG node and the parallel copy. An example of this transformation is shown in Figure 10.



(a) Original DDG (b) Transformed DDG

Figure 10: Downward and upward code motion is possible past cyclic parallel copies by renaming all operands of the respective instruction that also appear as operands of the copy.

Such a transformation is, by itself, not necessarily useful, ignoring indirect benefits that might arise in following optimization steps, e.g., instruction scheduling. However, the ability to move arbitrary instructions past a cyclic copy gives additional freedom and might be used to enable the elimination of a copy by one of the previously described techniques. Even more, it might sometimes be beneficial to turn a regular parallel copy into a cyclic one and move instructions past that copy in order to enable further possibilities for copy elimination.

3.4 Additional Remarks

The algorithms presented in the previous sections are invoked iteratively as long as parallel copies and candidate instructions, i.e., respective DDG nodes, exist that might be eligible for code motion, or a predefined threshold has been reached. So far, it was assumed that all instructions are amenable to renaming. However, in practice this is not always the case, in particular, when additional register constraints have to be accounted for. These constraints may arise from registers that are accessed by an instruction independent from its operands, i.e., *clobbered* registers, condition code registers, fixed operands. These registers can, of course, not be renamed. Another source of constraints arise from register usage conventions of the application binary interface (ABI), e.g., when function parameters are passed using registers on function calls. Renaming those registers is again not possible. These, and other forms of constraints, have not been discussed to simplify the presentation, but are relevant and have to be considered.

In addition, other forms of dependencies besides those discussed previously might appear in an actual DDG. The respective algorithms to detect cyclic dependencies and update the DDG have to be modified in order to preserve the program's original semantics throughout all code motion transformations. The algorithms presented in the previous sections mainly dealt with dependencies carrying registers in order to simplify the discussion.

A final remark concerns the elimination of empty parallel copies that were added to the DDG, e.g., during the splitting of cyclic parallel copies. It is not safe to simply remove the respective DDG nodes, since transitive dependencies might be lost. For instance, when the downward motion of a definition results in an empty parallel copy on the left-hand side having an incoming and outgoing output dependence labeled with the original destination register of the definition, an output dependence between the respective source and target node has to be added to the DDG when eliminating the empty copy. In practice, it is

preferable to avoid constructing empty copies and handle the respective corner cases directly during the DDG update. However, this will not be discussed in more detail in this work.

4 Experiments

We evaluate our approach for the ST2xx architecture within the production compiler (version 6.5.0) of STMicroelectronics, which is based on the Open64 optimizers[2] and the LAO [12] backend extension. The ST2xx architecture is a 4-way parallel VLIW architecture offering a single load/store unit, i.e., only a single memory access can be performed per cycle. The architecture defines 64 32-bit general purpose registers and 8 single-bit predicate registers, which can be used to control a conditional branch or a `select` operation that conditionally copies one if its two input operands to its output operand.

We applied our DDG-based copy elimination to the integer benchmarks of the SPEC2000 suite. The `252.eon` benchmark is omitted due to lacking C++ support. Also the `253.perlbmk` is not considered since certain system calls required by this benchmark are no longer supported by the ST2xx platform.

Register allocation is performed under SSA by a generic graph coloring register allocator featuring iterated coalescing [15], many copies are thus already eliminated. The remaining copies are induced by ϕ-operations introduced by the conversion to SSA form. The respective copies are converted to parallel copies, according to Definition 2, which are preliminarily placed *on* the CFG edges before the corresponding ϕ-operations. The parallel copies are then assigned to basic blocks using the *Edge Motion* technique of Bouchez et al. [3].

4.1 Copy Elimination after Full Coalescing

Our first setup shows the potential for our technique ($\texttt{DDG}_\phi$) after iterated coalescing, where coalescing of ϕ-related variables is enabled. The result is compared with a configuration ($\texttt{BASE}_\phi$), where only *Edge Motion* is performed, and

[2] `http://www.open64.net/`

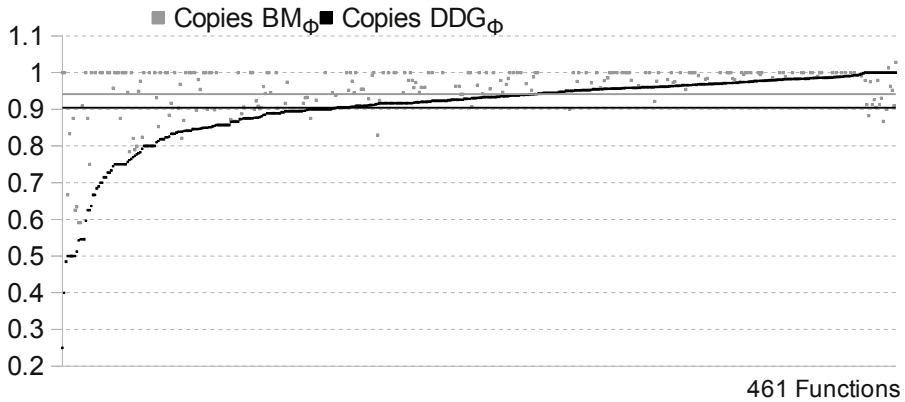


Figure 11: Remaining copies relative to the $\texttt{BASE}_\phi$ configuration, per function, after full coalescing (lower is better).

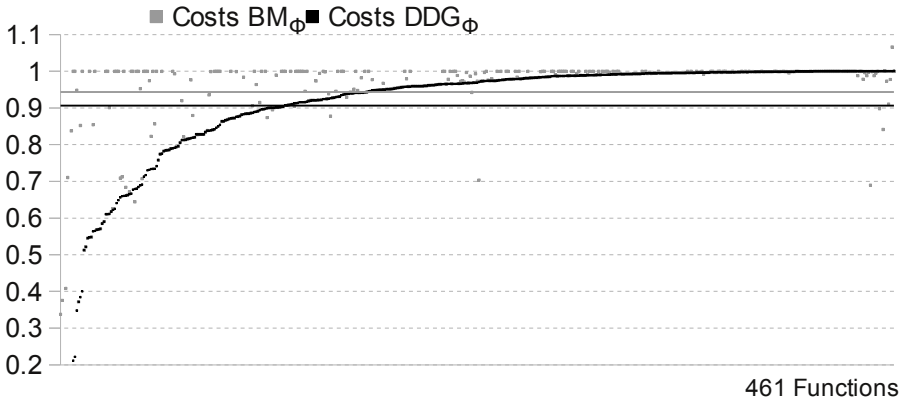


Figure 12: Remaining costs relative to the $\texttt{BASE}_\phi$ configuration, per function, after full coalescing (lower is better).

a configuration ($\texttt{BM}_\phi$), where *Block Motion* is performed in addition. Figure 11 compares the number of remaining copies, sorted ascending according to the $\texttt{DDG}_\phi$ configuration. The plot shows a data point for every function, where either $\texttt{BM}_\phi$ or $\texttt{DDG}_\phi$ eliminate some copies. In the best case, only 25% of the original copies remain for $\texttt{DDG}_\phi$ (`197.parser`), compared to 48% for $\texttt{BM}_\phi$ (`300.twolf`). On average over the 461 functions, only 90% of the copies remain for $\texttt{DDG}_\phi$, whereas for $\texttt{BM}_\phi$ 94% remain. Considering that iterated coalescing already delivers much better results than other heuristic coalescing techniques (about 20% in comparison to Brigg's conservative coalescing [15]), these results are surprisingly good. Due to the conversion between parallel copies and permutations, $\texttt{BM}_\phi$ may, in some cases, increase the number of copies. In the worst case, this increase amounts to 49% (`197.parser`), which is explained by an adverse interaction between *Block Motion* and inter-procedural register allocation.

Figure 12 shows the relative costs induced by register-to-register copies, i.e., the sum of the copies' estimated execution frequencies per function, sorted ascending with respect to the $\texttt{DDG}_\phi$ configuration, which again delivers consider-

	Number of Copies			Total Costs		
Benchmark	$\texttt{BASE}_\phi$	$\texttt{BM}_\phi$	$\texttt{DDG}_\phi$	$\texttt{BASE}_\phi$	$\texttt{BM}_\phi$	$\texttt{DDG}_\phi$
164.gzip	825	797	796	4212710	4211230	4211560
175.vpr	5437	5334	5312	3215486	3215023	3214308
176.gcc	40055	39075	38745	487753	475099	466556
181.mcf	228	227	223	1113	1106	1079
186.crafty	2149	2128	2122	247843	247839	247838
197.parser	3426	3352	3321	482189	363355	362490
254.gap	14446	14304	14129	16154931	16148202	16139236
255.vortex	22796	22722	22708	10922	10909	10898
256.bzip2	673	662	661	4362319	4361304	4361293
300.twolf	5201	4828	4752	1390838	1389218	1389016

Table 1: Total number of copies and total copy costs remaining for each benchmark after register allocation with full coalescing (lower is better).

ably better results. In the best case, only 0.8%(!) (`254.gap`) of the copy costs remain for $\texttt{DDG}_\phi$, whereas 7% (`176.gcc`) of the costs remain for $\texttt{BM}_\phi$. On average over the 461 functions, only 91% of the costs remain for our new approach, while for $\texttt{BM}_\phi$ 94% of the costs remain.

The number of remaining copies and their total costs over all 4811 functions is shown by Table 1. The trend observed previously is again reflected by these numbers, albeit at a reduced magnitude. On average, $\texttt{DDG}_\phi$ eliminates more than 3% of all the copies that could not be eliminated by iterated coalescing, whereas *Block Motion* eliminates just about 2%. For `300.twolf` both approaches perform best, eliminating 9% ($\texttt{DDG}_\phi$) respectively 7% ($\texttt{BM}_\phi$) of the copies. In terms of copy costs, `197.parser` shows the best reductions amounting to about 25% for both techniques.

4.2 Copy Elimination after *Decoupled* Register Allocation

In our second setup, we *mimic* decoupled, heuristic register allocation by deactivating the coalescing of ϕ-related variables, i.e., more parallel copies appear. The configurations (names without the ϕ subscript) remain unchanged otherwise. Figure 13 shows the remaining copies relative to the base configuration `BASE`. `DDG` performs considerably better, only 73% of the copies remain on average over the 2296 functions, where either `DDG` or `BM` are able to eliminate some copy. For the `BM` configuration, on the other hand, 82% of the copies remain. The medians for the DDG-based and block-motion-based configurations lie at 76% and 83% respectively. In the best case, `DDG` eliminates all copies (`181.mcf`, `186.crafty`, `197.parser`), while for `BM` 11% of the copies remain in the best case (`176.gcc`). As before, *Block Motion* increases the number of copies for certain functions, which amounts to 9% in the worst case (`176.gcc`).

The corresponding reductions in copy costs are shown by Figure 14. For `DDG` only 71% of the initial copy costs remain, while 81% of the costs remain for `BM`. Naturally, the copy costs for the functions where `DDG` is able to eliminate all copies become zero as well. The Block Motion algorithm is only able reduce the remaining copy costs to less than 1% for a single function (`186.crafty`).

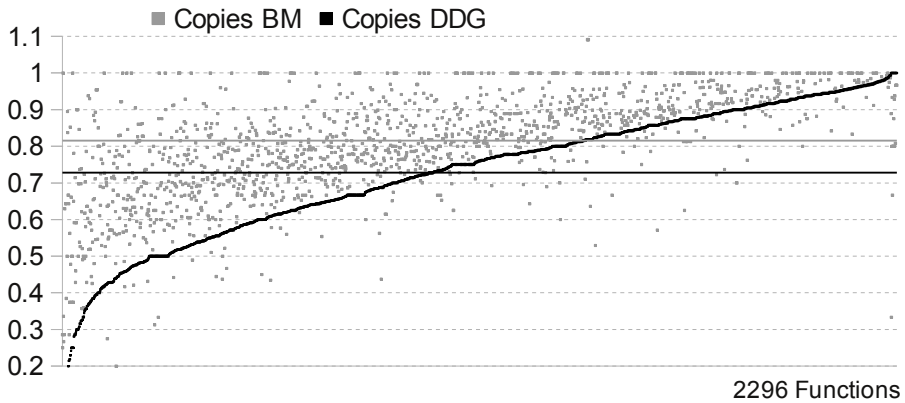


Figure 13: Remaining copies relative to the `BASE` configuration, per function, after *decoupled* register allocation (lower is better).

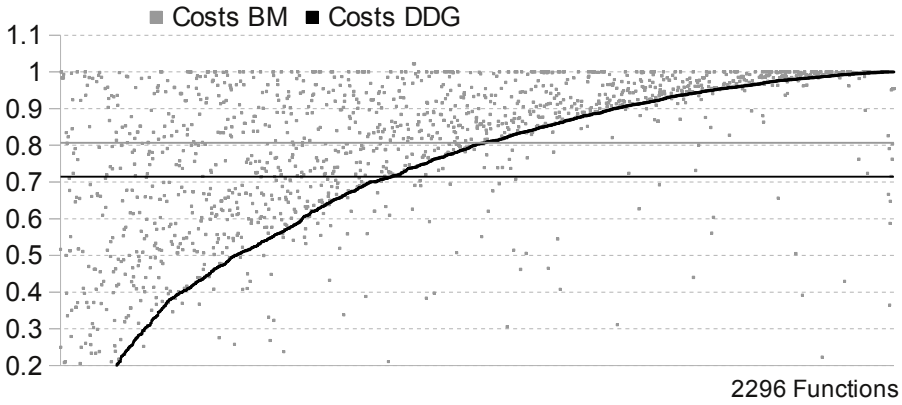


Figure 14: Remaining costs relative to the `BASE` configuration, per function, after *decoupled* register allocation (lower is better).

The medians lie at 80% and 88% respectively. For 6 functions of `176.gcc` *Block Motion* increases the copy costs between 2% and 22%.

A comparison of the total number of copies and their respective costs over all 4811 functions for the three configurations is given by Table 2. The DDG-based approach, on average, over all benchmarks eliminates 32% of all copies with respect to the `BASE` configuration. The best result is achieved for `164.gzip`, where 45% of the copies are eliminated. The approach performs even better with regard to copy costs, where the reduction amounts to 37% on average. The `254.gap` benchmark here shows the best result, 55% of the copy costs are eliminated. The configuration based on *Block Motion* shows better results than for our first setup, however, cannot reach the `DDG` configuration. On average over all benchmarks, 21% of the copies and 23% of the copy costs are eliminated. The best results are achieved for `300.twolf` and `197.parser` with reductions of 28% in the number of copies and 42% in the total copy costs respectively. Both configurations appear to have difficulties with the `255.vortex` benchmark, where `DDG` is able to eliminate only 13% and `BM` only 10% of the copies.

	Number of Copies			Total Costs		
Benchmark	`BASE`	`BM`	`DDG`	`BASE`	`BM`	`DDG`
164.gzip	3205	2392	1754	5357735	4290840	4255540
175.vpr	8614	7254	6570	7627027	6163193	5170911
176.gcc	64359	52862	47567	13382491	9736197	7300438
181.mcf	527	404	342	7212	5924	3549
186.crafty	5912	4306	3453	2693734	2134310	1284878
197.parser	6508	5353	4677	1039805	604656	578991
254.gap	30039	24932	20679	67426060	56894894	30334354
255.vortex	27881	25008	24175	15696	13514	12803
256.bzip2	1780	1347	1153	6683255	5120760	5075978
300.twolf	14640	10579	8812	15582390	11381541	9985798

Table 2: Total number of copies and total costs remaining for each benchmark after *decoupled* register allocation (lower is better).

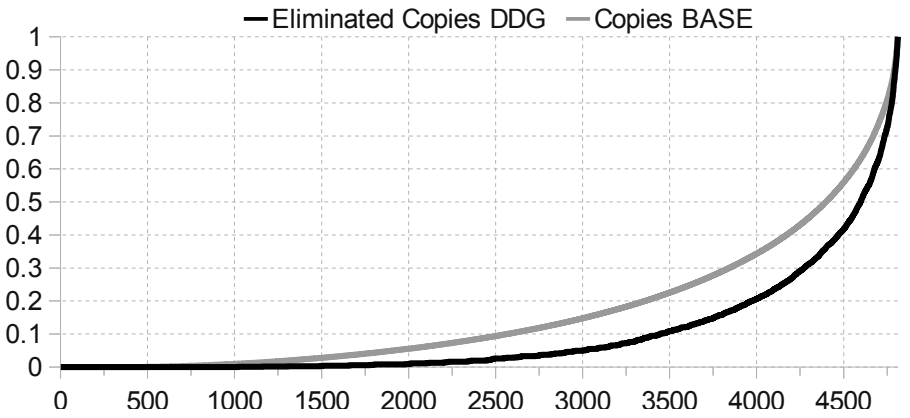


Figure 15: Accumulated and normalized number of copies eliminated by `DDG` in comparison to the accumulated and normalized number of copies for the `BASE` configuration, per function, after *decoupled* register allocation.

Note, however, that the total number of copies and their costs summarized by the table favors benchmarks with larger functions having more copies, which are often easy to eliminate. These large functions may dominate the overall numbers, as depicted by Figure 15. The figure shows the accumulated and normalized number of copies eliminated by `DDG` in comparison to the accumulated and normalized number of initial copies from `BASE`. For `BASE`, 414, i.e., less than 10%, out of the 4811 functions contain 50% of the total number of copies. During copy elimination this is even further amplified. For `DDG`, only 214 functions account for 50% of the eliminated copies. For the `BASE` configuration, these functions contain about 37% of the total number of copies of all functions.

4.3 Coalescing versus DDG-based Copy Elimination

Due to the conservative nature of iterated register coalescing we can compare the results of the two experimental setups presented in the previous sub-sections. Figure 16 shows the number of remaining copies per function for the `BASE` and `DDG` configurations (after decoupled register allocation) in relation to the $\texttt{BASE}_\phi$ configuration, which performs full coalescing. Disabling the coalescing of ϕ-related variables (`BASE`) leads to a dramatic increase in the number of copies by 87% on average, per function. While 1812 (38%) of the functions do not show any significant increase, 293 (18%) show an increase by a factor of two or more – up to a factor of 49 in the worst case.

Given the local scope of our DDG-based copy elimination, it turns out to be surprisingly effective. On average, the increase for `DDG` in comparison to $\texttt{BASE}_\phi$ amounts to only 37% per function, i.e., our technique is able to make up for more than half of the losses. Almost half of the functions, 2154 (42%) contain the same amount or less copies than with full coalescing enabled, and only 357 (7%) show increases of a factor of two or more. The worst case increase is also reduced down to a (still high) factor of 24 – only 5 out of all functions show an increase larger than 16 and 27 functions an increase larger than 8.

The results per benchmark follow this trend, as depicted by Figure 17. The `255.vortex` benchmark is the least impacted by the disabled coalescing of

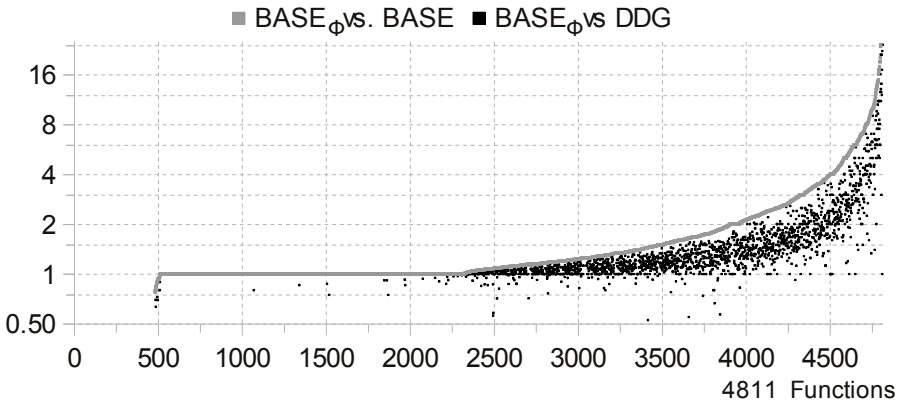


Figure 16: Increase in the number of copies relative to the base configuration `BASE`$_\phi$, per function (lower is better).

ϕ-related variables, showing increases of 10% and 6% for `BASE` and `DDG` respectively. The worst results are observed for `164.gzip`, with an increase of a factor of 4 and 2.13 for these two configurations respectively. On average, we thus observe an increase of a factor of 2.28 for the base configuration, compared to an increase of 49% for our DDG-based technique. The standard Block Motion approach consistently gives results worse than our new method. The increases in copy costs follow similar trends as the total number of copies.

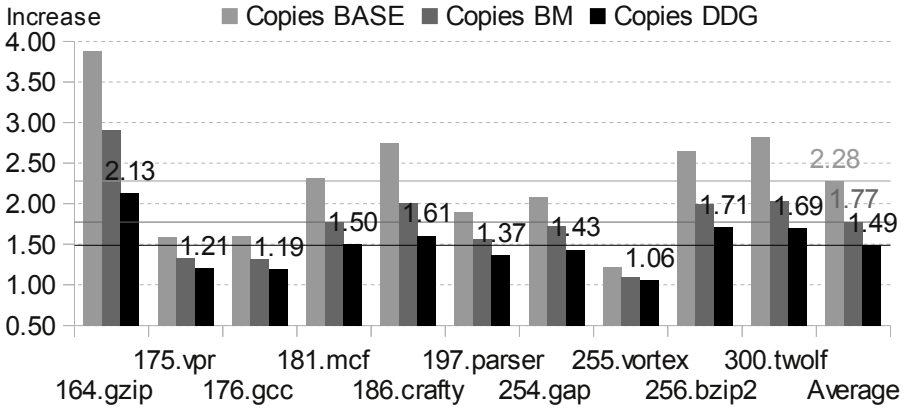


Figure 17: Increase in the number of copies relative to the base configuration `BASE`$_\phi$ (lower is better).

5 Related Work

A standard approach to copy elimination during register allocation is register coalescing [10]. The problem, however, is that coalescing might increase the register pressure and may lead to additional spilling. Heuristics thus try to find a good balance [7, 15, 20] between overly aggressive coalescing and spilling. Bouchez et al. showed that various variants of the coalescing problem are NP-complete [4]. Hack and Grund [16] proposed an optimal solution to the coalescing problem based on integer linear programming. Our approach does not require the construction of interference graphs and is thus better suited for decoupled approaches.

Similar to Brisk et al. [8], Hack et al. proposed techniques to eliminate copies during register assignment [6]. The basic idea is to bias the assignment such that copy-related variables are assigned the same register. In another approach, proposed by Buchwald et al. [9], the register assignment is modeled as a non-linear optimization problem (PBQP [13]), that can be solved heuristically or optimally using branch-and-bound. These approaches are complementary to our work, it thus might prove interesting to combine their respective strengths: the register assignment optimizes *global*, ϕ-related live-ranges spanning multiple basic blocks, while the DDG-based copy elimination handles basic-block-local assignment mismatches.

As proposed in this work, local recoloring after the assignment can be performed. Hack and Goos proposed a recoloring technique on interference graphs [17] that tries to fix-up mismatches locally, which are then propagated throughout the entire IG. Parallel Copy Motion [3], briefly introduced in Section 2, aims at finding a good placement of copies in the control flow graph and within basic blocks. The main advantage of this technique, similar to our technique, is that the construction of an IG is avoided. It is thus best suited for dynamic code generators, such as just-in-time compilers.

In contrast to this work, none of the previous approaches exploits the reordering of instructions to eliminate copies.

6 Conclusion

This work presents a new algorithm to eliminate register-to-register copies after register allocation based on the idea of local recoloring. Our approach operates on data dependence graphs and thus has the unique ability to reorder instructions, if this appears to be profitable.

Our experiments show that even after traditional copy elimination, using a state-of-the-art coalescing algorithm, our approach is able to eliminate additional copies. The approach also proves very powerful as an alternative to coalescing in the context of decoupled register allocation. In both settings, our DDG-based algorithm offers considerable improvements with respect to Parallel Copy Motion, a previously proposed recoloring technique [3].

A limitation of our approach, in comparison to traditional coalescing, is the local scope, i.e., we currently limit our approach to basic blocks. It should be fairly easy to extend the presented algorithms to operate on data dependence graphs of extended basic blocks, super blocks, or even traces [19, 14]. The increased scope of the optimization might then improve its effectiveness – in particular with respect to copy costs. It might also be interesting to investigate a closer intertwining between *Edge Motion* and our technique. For instance, it might be profitable to process the basic blocks in post-order using our technique, while propagating the remaining copies to neighboring basic blocks along control-flow edges.

We could, furthermore, exploit copies more aggressively after copy elimination in the final instruction scheduling pass. Uses of a register can be scheduled freely before or after a related copy involving the same register, if no dependencies exist between the use and the copy. The scheduler might even introduce copies to avoid costly stalls. It might thus be interesting to extend our approach to operate in concert with instruction scheduling.

References

[1] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *Proc. of the Conf. on Programming Language Design and Implementation*, PLDI '01, pages 243–253. ACM, 2001.

[2] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, September 2009.

[3] Florent Bouchez, Quentin Colombet, Alain Darte, Fabrice Rastello, and Christophe Guillon. Parallel copy motion. In *Proc. of the Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 1:1–1:10. ACM, 2010.

[4] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *Proc. of the Symposium on Code Generation and Optimization*, CGO '07, pages 102–114. IEEE, 2007.

[5] Matthias Braun and Sebastian Hack. Register spilling and live-range splitting for SSA-form programs. In *Proc. of the Conf. on Compiler Construction*, CC '09, pages 174–189. Springer, 2009.

[6] Matthias Braun, Christoph Mallon, and Sebastian Hack. Preference-guided register assignment. In *Proc. of the Conf. on Compiler Construction*, CC '10, pages 205–223. Springer, 2010.

[7] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16:428–455, 1994.

[8] Philip Brisk, Ajay Kumar Verma, and Paolo Ienne. An optimistic and conservative register assignment heuristic for chordal graphs. In *Proc. of the Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 209–217. ACM, 2007.

[9] Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch. SSA-based register allocation with PBQP. In *Proc. of the Conf. on Compiler Construction*, pages 42–61. Springer, 2010.

[10] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. of the Symposium on Compiler Construction*, pages 98–105. ACM, 1982.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, 1991.

[12] B. Dupont de Dinechin, F. de Ferrière, C. Guillon, and A. Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *Proc. of the Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 93–102. ACM, 2000.

[13] Erik Eckstein. *Code Optimization for Digital Signal Processors*. PhD thesis, Institut für Computersprachen, Technische Universität Wien, November 2003.

[14] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[15] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18:300–324, 1996.

[16] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *Proc. of the Conf. on Compiler Construction*, CC'07, pages 111–125. Springer, 2007.

[17] Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. In *Proc. of the Conf. on Programming Language Design and Implementation*, PLDI '08, pages 227–237. ACM, 2008.

[18] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *Proc. of the Conf. on Compiler Construction*, CC '06, pages 247–262. Springer, 2006.

[19] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, 1993.

[20] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26:735–765, 2004.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399